Вестник Донского государственного технического университета. 2019. Т. 19, № 3. С. 290–300. ISSN 1992-5980 eISSN 1992-6006
Vestnik of Don State Technical University. 2019. Vol. 19, no. 3, pp. 290–300. ISSN 1992-5980 eISSN 1992-6006

ИНФОРМАТИКА, ВЫЧИСЛИТЕЛЬНАЯ ТЕХНИКА И УПРАВЛЕНИЕ
INFORMATION TECHNOLOGY, COMPUTER SCIENCE, AND MANAGEMENT



УДК 519.689 https://doi.org/10.23947/1992-5980-2019-19-3-290-300

Ускоренный препроцессинг в задаче поиска подстрок в строке[*]

А. В. Мазуренко[1], Н. В. Болдырихин[2]**
[1] ООО «ДДОС-Гвард», г. Ростов-на-Дону, Российская Федерация
[2] Донской государственный технический университет, г. Ростов-на-Дону, Российская Федерация

Accelerated preprocessing in task of searching substrings in a string [***]

A. V. Mazurenko[1], N. V. Boldyrikhin[2]**
[1] DDoS-GUARD LLC, Rostov-on-Don, Russian Federation
[2] Don State Technical University, Rostov-on-Don, Russian Federation

Введение. Бурное развитие таких систем, как Yandex, Google и пр., предопределило актуальность задачи поиска подстрок в строке. На сегодняшний день активно исследуются подходы к ее решению. Эта задача используется при создании систем управления базами данных, поддерживающих ассоциативный поиск. Кроме того, она применима при решении вопросов информационной безопасности, создании антивирусных программ. Алгоритмы поиска подстрок в строке используются в задачах обнаружения, основанного на сигнатурах.
Материалы и методы. Решение задачи базируется на алгоритме Ахо — Корасик, который представляет собой классический способ осуществления поиска подстрок в строке. Вместе с тем применен новый подход в части, касающейся предварительной обработки.
Результаты исследования. Показана возможность построения функции перехода и суффиксных ссылок при помощи суффиксных массивов и специальных отображений. Исследована взаимосвязь между префиксным деревом и суффиксными массивами. Это дало возможность разработать принципиально новый способ построения функций перехода и ошибок.
Полученные результаты позволяют существенно сократить время, затрачиваемое на предвыборную обработку множества строк образцов при использовании целочисленного алфавита.
В статье приведено восемь алгоритмов. Оценены разработанные алгоритмы. Полученные результаты сопоставлены с ранее известными. Доказаны две теоремы и восемь лемм. Приведены два примера, иллюстрирующие особенности практического применения разработанной процедуры препроцессинга.
Обсуждение и заключения. Предложенная в данной статье процедура препроцессинга основывается на связи между суффиксным массивом, созданным на основе множества строк образцов, и построением функций перехода и ошибок на начальных этапах работы алгоритма Ахо — Корасик. Такой подход отличен от традиционного и требует

Introduction. A rapid development of the systems such as Yandex, Google, etc., has predetermined the relevance of the task of searching substrings in a string, and approaches to its solution are actively investigated. This task is used to create database management systems that support associative search. Besides, it is applicable in solving information security issues and creating antivirus programs. Algorithms of searching substring in a string are used in signature-based discovery tasks.
Materials and Methods. The solution to the problem is based on the Aho-Corasick algorithm which is a typical technique of searching substrings in a string. At the same time, a new approach regarding preprocessing is employed.
Research Results. The possibility of constructing the transition function and suffix references through suffix arrays and special mappings, is shown. The relationship between the prefix tree and suffix arrays was investigated, which provided the development of a fundamentally new method of constructing the transition and error functions. The results obtained enable to substantially shorten the time intervals spent on the preelection processing of a set of pattern strings when using an integer alphabet. The paper lists eight algorithms. The developed algorithms are evaluated. The results obtained are compared to the formerly known. Two theorems and eight lemmas are proved. Two examples illustrating features of the practical application of the developed preprocessing procedure are given.
Discussion and Conclusions. The preprocessing procedure proposed in this paper is based on the communication between the suffix array built on the ground of a set of pattern strings and the construction of transition and error functions at the initial stages of the Aho-Corasick algorithm. This approach differs from the traditional one and requires the use of algo-



[*] Работа выполнена в рамках инициативной НИР.
[**] E-mail: mazurencoal@gmail.com, boldyrikhin@mail.ru
[***] The research is done within the frame of the independent R&D.

использования алгоритмов, позволяющих построить суффиксный массив за линейное время. Таким образом, описаны алгоритмы, позволяющие существенно сократить время на предварительную обработку множества строк образцов при условии использования определенного типа алфавита по сравнению с известным подходом, предложенным А. Ахо и М. Корасик.
Результаты исследований, приведенные в статье, могут быть применены в антивирусных программах, использующих поиск сигнатур вредоносных информационных объектов в памяти вычислительной системы. Кроме того, данный подход к решению задачи поиска подстроки в строке позволяет значительно ускорить работу систем управления баз данных, применяющих ассоциативный поиск.

rithms providing a suffix array in linear time. Thus, the algorithms that enable to significantly reduce the time for preprocessing of a set of pattern strings under the condition of using a certain type of alphabet in comparison to the known approach proposed in the Aho- Corasick algorithm are described. The research results presented in the paper can be used in antivirus programs that apply searching for signatures of malicious data objects in the memory of a computer system. In addition, this approach to solving the problem on searching substrings in a string will significantly speed up the operation of database management systems using associative search.

Ключевые слова: поиск подстроки, алгоритм Ахо — Корасик, префиксное дерево, суффиксный массив, поиск информации, функция ошибок, функция перехода.

Keywords: string searching, Aho-Corasick algorithm, prefix tree, suffix array, information search, error function, transition function

Образец для цитирования: Мазуренко, А.В. Ускоренный препроцессинг в задаче поиска подстрок в строке / А. В. Мазуренко Н. В. Болдырихин // Вест- ник Дон. гос. техн. ун-та. — 2019. — Т. 19, № 3. — С. 290‒300. https://doi.org/ 10.23947/1992-5980-2019-19-3- 290-300

For citation: A.V. Mazurenko, N.V.Boldyrikhin. Accelerated preprocessing in task of searching substrings in a string. Vestnik of DSTU, 2019, vol. 19, no. 3, pp. 290‒300. https://doi.org/10.23947/1992-5980-2019-19-3-290-300

Введение. В настоящее время существенно возрастает важность обеспечения кибербезопасности распределенных информационных систем и отдельных вычислительных средств [1]. Диапазон таких задач достаточно широк [1–10]. Особое место занимают вопросы создания эффективного антивирусного программного обеспечения (ПО). Одной из важнейших задач, решаемых таким ПО, является поиск подстрок в строке [1, 5, 6, 10–13].

Материалы и методы. Задача поиска подстрок заключается в нахождении всех строк в тексте *T* общей длиной *m*, совпадающих с каким-либо образцом из заданного множества образцов *P*. Положим, сумма длин всех элементов *P*, состоящих из символов алфавита *I*, равна *n*. Решение этой задачи было предложено А. Ахо и М. Корасик [6, 10]. В их алгоритме время предвыборной обработки составляет $O(n|I|)$, а время поиска — $O(m|I|+k)$. Здесь *k* — количество совпадений, найденных в тексте со строками, принадлежащими множеству образцов.

В настоящее время задача поиска подстроки в строке активно исследуется по двум причинам:
— поисковые системы активно развиваются [11];
— процесс обнаружения в антивирусных программных продуктах основывается на сигнатурах [1].

В этой связи были созданы алгоритмы, которые приходится выбирать в зависимости от определенных потребностей пользователя. Новейшие результаты, полученные при решении задачи поиска множества подстрок, описаны в работе [13].

Представленные в данной статье результаты основываются на связи между суффиксным массивом, созданным на основе множества строк образцов, и построением функций перехода и ошибок на начальных этапах работы алгоритма Ахо — Корасик. Такой подход отличен от традиционного и требует использования алгоритмов, позволяющих построить суффиксный массив за линейное время. Итак, в статье описаны алгоритмы, с помощью которых время выполнения предвыборной обработки сокращается до $O(n)$.

Пусть задан алфавит *I*, множество образцов $P=\{P_1,P_2,...,P_k\}$, где $P_i \in I^*$, $i=\overline{1,k}$. Обозначим через $n=\sum_{i=1}^{k}|P_i|$. Будем считать, что алфавит *I* представляет собой ограниченный промежуток целых чисел. Граница может зависеть от длины рассматриваемой строки $s \in I^*$ либо представлять собой промежуток $[0,c]$, где *c* — натуральное число: $c \geq |s|$. Пусть $\varepsilon \in I$ — пустая строка.

Пусть *goto* — функция перехода, а *failure* — функция ошибок. Приводимые модификации касаются методов построения упомянутых функций, используемых в алгоритме Ахо — Корасик [6, 10].

Положим, $SuffArr(s)$ — некоторый алгоритм построения суффиксного массива для строки $s \in I^*$ за линейное время. Описание таких алгоритмов можно найти, например, в [12–15].

Пусть $x, y \in I^*$. Тогда, $lcp(x, y)$ — наибольший общий префикс строк *x* и *y*.

Рассмотрим строку $s \in I^*$, $s = s\left[s[0]s[1]...s[n-1]\right]$. Пусть $s\left[s[i]s[i+1]...s[j]\right]$ — подстрока *s*, включающая символы от *i* до *j*, где $i \le j$, $i, j = \overline{0, n-1}$. Обозначим через p_s суффиксный массив, соответствующий строке *s*. Пусть

$$p_s = p_s\left[p_s[0]p_s[1]...p_s[n-1]\right],$$

то есть $s\left[s\left[p_s[0]\right]...s\left[n-1\right]\right] < s\left[s\left[p_s[1]\right]...s\left[n-1\right]\right] < ... < s\left[s\left[p_s[n-1]\right]...s\left[n-1\right]\right]$.

Для построения суффиксного массива будет использован алгоритм, описанный в [15].

Положим, $\alpha_i \notin I$, $\alpha_i \ne \alpha_j$, $1 \le i < j \le k+1$, $\alpha_1 < \alpha_2 < ... < \alpha_{k+1}$. Пусть $\forall b \in I \quad \alpha_i < b$, где $1 \le i \le k+1$. Допустим, что $P \ne \varnothing$, $alpha = \{\alpha_1, \alpha_2, \ldots, \alpha_k, \alpha_{k+1}\}$.

Алгоритм обработки суффиксного массива p_s

Здесь $s \in I^*$: $s = \alpha_1 P_1 \alpha_2 P_2 \ldots \alpha_k P_k \alpha_{k+1}$, $P_i \in I^*$, $1 \le i \le k$.

$Adaptation\left(s, p_s, alpha\right)$

1. $new_array \leftarrow \varepsilon$
2. $for\ \left(i \leftarrow |alpha|; i < |s|; i{+}{+}\right)$ {
3. $j \leftarrow 0$
4. $while\ \left(s\left[s[p_s[i]]...s[|s|-1]\right]\left[j\right] \notin alpha\right)$ {
5. $new_array\left[i\right]\left[j\right] \leftarrow s\left[s[p_s[i]]...s[|s|-1]\right]\left[j\right]$
6. $j \leftarrow j + 1$
7. }
8. }
9. $ordered_list\left[0\right] \leftarrow new_array\left[0\right]$
10. $for\ \left(i \leftarrow 1; i < |s| - |alpha|; i{+}{+}\right)$ {
11. $j \leftarrow 0$
12. $if\ \left(new_array\left[i\right] \ne new_array\left[i-1\right]\right)$ {
13. $ordered_list\left[i\right] \leftarrow new_array\left[i\right]$
14. $j \leftarrow j + 1$
15. }
16. }
17. *return ordered_list*

Лемма 1. Пусть $P = \{P_1, P_2, \ldots, P_k\}$, $s = \alpha_1 P_1 \alpha_2 P_2 \ldots \alpha_k P_k \alpha_{k+1}$. Тогда алгоритм *Adaptation* строит массив лексикографически упорядоченных суффиксов образцов, принадлежащих *P*, за время $O\left(|s| - |alpha|\right)$.

Доказательство. В цикле 2–8 выполняется построение массива *new_array*, *i*-й элемент которого представляет собой такой префикс соответствующего суффикса *s*, который включает в себя все символы данного суффикса, начиная с нулевой позиции до его первого элемента, принадлежащего множеству *alpha*. При этом при помощи суффиксного массива p_s перебираются все суффиксы *s* согласно их лексикографическому порядку. Таким образом, массив *new_array* состоит из всех суффиксов образцов, принадлежащих *P* согласно их лексикографическому упорядочиванию, причем возможно повторение некоторых суффиксов. Заметим, что из рассмотрения исключаются все строки, начинающиеся с символов, принадлежащих массиву *alpha*, то есть первые $|alpha|$ суффиксы. Далее в цикле 10–16 при помощи массива *new_array* строится массив *ordered_list* путем исключения повторяющихся строк. Для этого в силу лексикографического порядка следования строк достаточно проверить, совпадает ли рассматриваемая строка с предыдущей.

Цикл 2–8 выполняется за время $O(|s|-|alpha|)$, поскольку из рассмотрения исключаются все строки, начинающиеся с символов, принадлежащих массиву *alpha*. В цикле 10–16 происходит $|s|-|alpha|$ сравнений строк. Таким образом, получаем асимптотическую оценку времени работы алгоритма $O(|s|-|alpha|)$. Лемма доказана.

Алгоритм разбиения согласно лексикографическому упорядочению

Здесь s — массив лексикографически упорядоченных строк.

$DandC(s)$

1. $sub[0] \leftarrow 0$
2. $j \leftarrow 0$
3. $for\ (i \leftarrow 0; i < |s|-1; i++)$ {
4. $if\ (s[i] \neq s[i+1])$ {
5. $\mathrm{sub}[j] \leftarrow i+1$
6. $j \leftarrow j+1$
7. }
8. $sub[j] \leftarrow |s|$
9. *return sub*

Лемма 2. Алгоритм *DandC* на основе массива лексикографически упорядоченных строк *s* строит массив *sub*, состоящий из целых положительных чисел, указывающих индексы, соответствующие первым строкам среди строк, у которых равны первые символы, за время $O(|s|)$.

Доказательство. Граница, соответствующая первому символу, начинается с 0, что соответствует присваиванию, выполняемому на шаге 1. В цикле 3–7 последовательно сравниваются первые символы *i*-й и (*i* + 1)-й строк, где $i = \overline{0,|s|-2}$. Если символы не равны, то в массив *sub* записывается начало границы, соответствующей следующему символу. В противном случае выполнение цикла продолжается. Правая граница последнего символа соответствует количеству строк в массиве *s* (шаг 8).

Сравнение на шаге 4 происходит за время $O(1)$, как и запись на шаге 5 и инкрементирование на шаге 6. Таким образом, цикл 3–7 выполняется за время $O(|s|)$. Лемма доказана.

Алгоритм построения первой связи

Здесь *tree* — дерево, $lex_words \in I^*$, $link_num$ — номер некоторого символа строки lex_words, v — порядковый номер нового узла, который присоединяется к узлу с порядковым номером $node_number$.

$BuildFirstLink(tree\&, lex_words\&, v\&, link_num, node_number)$

1. $new\ tree.node[v]$
2. $tree.node[v].state \leftarrow lex_words[lex_words[0]...lex_words[link]]$
3. $new\ tree.node[node_number].link \leftarrow tree.node[v]$
4. $tree.node[node_number].link.symbol \leftarrow lex_words[link_num]$
5. $v \leftarrow v+1$

Лемма 3. Алгоритм *BuildFirstLink* осуществляет построение нового узла с порядковым номером *v* и дуги, ведущей от узла *node_number* к новому узлу *v*, в дереве *tree* за время $O(1)$.

Алгоритм построения связи с подстрокой

Здесь *tree* — дерево, $lex_words \in I^*$, v — порядковый номер нового узла, который присоединяется к узлу с порядковым номером $start$.

$BuildSubstringLink(tree\&, lex_words\&, v\&, start)$

1. $for\ (k \leftarrow start; k < |lex_words|; k++)$ {

2. $new\ tree.node[v]$

3. $tree.node[v].state \leftarrow lex_words[lex_words[0]...lex_words[k]]$

4. $new\ tree.node[v-1].link \leftarrow tree.node[v]$

5. $tree.node[v-1].link.symbol \leftarrow lex_words[k]$

6. $v \leftarrow v+1$

7. }

Лемма 4. Алгоритм *BuildSubstringLink* осуществляет построение новых узлов в дереве *tree*, соответствующих всем префиксам строки *lex_words*, начиная с префикса $lex_words[lex_words[0]...lex_words[start]]$ за время $O(|lex_words|-start)$.

Алгоритм построения последней связи

Здесь *tree* — дерево, $lex_words \in I^*$, v — порядковый номер новой дуги, I — алфавит.

$BuildLastLink(tree\&, lex_words, v, I)$

1. $new\ tree.node[0].link[v] \leftarrow tree.node[0]$

2. $symbols \leftarrow \varnothing$

3. $for\ (i \leftarrow 0; i < |lex_words|; i{+}{+})$ {

4. $symbols[i] \leftarrow lex_words[i][0]$

5. $j \leftarrow 0$

6. $for\ (i \leftarrow 0; i < |I|; i{+}{+})$ {

7. if $(I[i] \notin symbols)$ {

8. $tree.node[0].link[v].symbol[j] \leftarrow I[i]$

9. $j \leftarrow j+1$

10. }

11. }

Лемма 5. Алгоритм *BuildLastLink* строит при корневом узле петлю. Ее пометка соответствует множеству символов, по которым невозможно перейти в другие узлы дерева *tree* из корневого узла за время $O(|lex_words|+|I|)$.

Алгоритм построения переходов

Здесь lex_words — массив лексикографически упорядоченных строк.

$CreateLink(lex_words)$

1. $str \leftarrow \varnothing$

2. $sub \leftarrow \mathrm{DandC}(lex_words)$

3. $v \leftarrow 1$

4. $tree \leftarrow \varnothing$

5. $tree.node[0].state \leftarrow \varepsilon$

6. $for\ (i \leftarrow 0; i < |sub|-1; i{+}{+})$ {

7. $BuildFirstLink(tree, lex_words[sub[i]], v, 0, 0)$

8. $BuildSubstringLink(tree, lex_words[sub[i]], v, 1)$

9. $for\ (j \leftarrow sub[i]+1; j < sub[i+1]; j{+}{+})$ {

10. $temp \leftarrow |lcp(lex_words[j-1], lex_words[j])|+1$

11. $z \leftarrow tree.getStateNumber(lcp(lex_words[j-1], lex_words[j]))$

12. $BuildFirstLink(tree, lex_words[j], v, temp, z)$

13. $BuildSubstringLink\left(tree, lex_words\left[j\right], v, temp\right)$

14. }

15. }

16. $BuildLastLink\left(tree, lex_words, v, lex_words\right)$

17. *return tree*

Лемма 6. Алгоритм *CreateLink* осуществляет построение префиксного дерева *tree* с петлей при корневом узле за время $O\left(\sum_{i=0}^{|lex_words|-1}|lex_words[i]|\right)$.

Доказательство. На шаге 2 выполняется алгоритм *DandC* (см. лемму 2), после чего на шаге 5 создается корневой узел с порядковым номером 0 дерева *tree*. Его состояние полагается равным пустой строке ε. Рассмотрим цикл 6–15 на *i*-м шаге работы.

На шаге 7 при помощи алгоритма *BuildFirstLink* создается узел, состояние которого соответствуют первому символу строки $lex_words\left[sub[i]\right]$. Учитывая построение массива *sub*, можно утверждать, что такой символ прежде не встречался среди первых символов предыдущих строк. Тогда на шаге 8 реализация алгоритма *BuildSubstringLink* последовательно создает узлы, состояние которых соответствует всем префиксам строки $lex_words\left[sub[i]\right]$, исключая префикс, построенный в предшествующей строке.

В цикле 9–14 при помощи алгоритмов *BuildFirstLink* и *BuildSubstringLink* выполняем те же действия со строками, лежащими в целочисленном промежутке $\left[sub[i]+1, sub[i+1]-1\right]$. Поскольку у каждой такой строки есть общий ненулевой префикс с предыдущей строкой, то алгоритм немедленно переходит в состояние, соответствующее наибольшему общему префиксу, начиная с которого необходимо строить новые узлы. На шаге 16 при помощи алгоритма *BuildLastLink* создается петля при корневом узле.

Шаги 12 и 13 выполняются за время

$$O(1)+O\left(\left|lex_words\left[j\right]\right|-\left|lcp\left(lex_words\left[j-1\right], lex_words\left[j\right]\right)\right|-1\right)=$$
$$=O\left(\left|lex_words\left[j\right]\right|-\left|lcp\left(lex_words\left[j-1\right], lex_words\left[j\right]\right)\right|\right).$$

Таким образом, из лемм 2, 3 и 4 следует, что цикл 9–14 выполняется за время

$$O\left(\sum_{j=sub[i]+1}^{sub[i+1]-1}\left|lex_words\left[j\right]\right|-\left|lcp\left(lex_words\left[j-1\right], lex_words\left[j\right]\right)\right|\right).$$

Цикл 6–14 выполняется за время

$$O\left(\sum_{i=0}^{|sub|-2}\left|lex_words\left[sub[i]\right]\right|\right)+$$
$$+O\left(\sum_{i=0}^{|sub|-2}\sum_{j=sub[i]+1}^{sub[i+1]-1}\left|lex_words\left[j\right]\right|-\left|lcp\left(lex_words\left[j-1\right], lex_words\left[j\right]\right)\right|\right)=$$
$$=O\left(\sum_{i=0}^{|sub|-1}\left|lex_words\left[sub[i]\right]\right|-\left|lex_words\right|\right).$$

Из леммы 5 следует, что шаг 16 выполняется за время $O\left(\left|lex_words\right|+\sum_{i=0}^{|lex_words|-1}\left|lex_words\left[i\right]\right|\right)$.

Таким образом, получаем асимптотическую оценку времени работы алгоритма

$$O\left(\left|lex_words\right|\right)+O\left(\sum_{i=0}^{|sub|-1}\left|lex_words\left[sub[i]\right]\right|-\left|lex_words\right|\right)+$$
$$+O\left(\left|lex_words\right|+\sum_{i=0}^{|lex_words|-1}\left|lex_words\left[i\right]\right|\right)=O\left(\sum_{i=0}^{|lex_words|-1}\left|lex_words\left[i\right]\right|\right).$$

Лемма доказана.

Алгоритм построения функции *goto*

Здесь P — множество строк образцов.

$ConstructGoto\left(P\right)$

1. $s \leftarrow \alpha_1 P_1 \alpha_2 P_2 \ldots \alpha_k P_k \alpha_{k+1}$

2. $p_s \leftarrow SuffArr(s)$
3. $alpha \leftarrow \{\alpha_1, \alpha_2, \ldots, \alpha_k, \alpha_{k+1}\}$
4. $ordered_list \leftarrow Adaptation(s, p_s, alpha)$
5. $j \leftarrow 0$
6. $lex_words \leftarrow \varnothing$
7. $P_length \leftarrow \{|P_1|, |P_2|, \ldots, |P_k|\}$
8. $for\ (i \leftarrow 0; i < |ordered_list|; i{+}{+})$
9. $if\ ((|ordered_list[i]| \in P_length)\ and\ (ordered_list[i] \in P))$
10. $lex_words[j{+}{+}] \leftarrow ordered_list[i]$
11. $goto \leftarrow CreateLink(lex_words)$
12. $return\ goto$

Напомним, что $n = \sum_{i=1}^{k} |P_i|$ для множества образцов $P = \{P_1, P_2, \ldots, P_k\}$.

Теорема 1. Алгоритм *ConstructGoto* строит функцию *goto* за время $O(n)$.

Доказательство. На шаге 2 строится суффиксный массив p_s для строки *s*. На шаге 4 при помощи алгоритма *Adaptation* в массив *ordered_list* записываются все суффиксы строк, принадлежащих множеству образцов *P*. При этом не исключены повторения. В цикле 8–10 строится массив *lex_words*, содержащий суффиксы, принадлежащие *P*, расположенные в лексикографическом порядке без повторений. На шаге 11 происходит построение префиксного дерева с петлей при корневом узле на основе строк, которые содержатся в массиве *lex_words*. Структура данных, возвращаемая алгоритмом *CreateLink*, в точности задает функцию *goto*.

Шаг 2 выполняется за время $O(n+k+1)$ [12]. Из леммы 1 следует, что шаг 4 выполняется за время $O(n+k+1-k-1) = O(n)$. В цикле 8–10 рассматриваются только строки, длина которых равна длине какого-либо образца.

Таким образом, потребуется не более $O(n)$ проверок для нахождения образцов *P*. Из леммы 6 следует, что шаг 11 выполняется за время $O\left(\sum_{i=0}^{|lex_words|-1} |lex_words[i]|\right) = O(n)$. Так как $k \le n$, получаем асимптотическую оценку времени работы алгоритма $O(n) + O(n+k+1) = O(n)$. Теорема доказана.

Результаты исследования

Пример 1.

Пусть $P = \{one, on, once, cell, lull, eye, near\}$. Тогда

$$s = \alpha_1 one \alpha_2 on \alpha_3 once \alpha_4 cell \alpha_5 lull \alpha_6 eye \alpha_7 near \alpha_8. \quad (1)$$

В табл. 1 представлен результат работы алгоритма построения функции *goto* при поступлении на вход строки *s* (1).

Таблица 1

Структура префиксного дерева

Номер *node*	Состояние *node*	Состояния, в которые ведут ветви *link* от *node*	Символы на ветвях *link* от *node*
0	ε	1. *c*; 2. *e*; 3. *l*; 4. *n*; 5. *o*	1. *c*; 2. *e*; 3. *l*; 4. *n*; 5. *o*
1	*c*	1. *ce*	1. *e*
2	*ce*	1. *cel*	1. *l*
3	*cel*	1. *cell*	1. *l*
4	*cell*	–	–
5	*e*	1. *ey*	1. *y*
6	*ey*	1. *eye*	1. *e*
7	*eye*	–	–
8	*l*	1. *lu*	1. *u*

Номер *node*	Состояние *node*	Состояния, в которые ведут ветви *link* от *node*	Символы на ветвях *link* от *node*
9	*lu*	1. *lul*	1. *l*
10	*lul*	1. *lull*	1. *l*
11	*lull*	–	–
12	*n*	1. *ne*	1. *e*
13	*ne*	1. *nea*	1. *a*
14	*nea*	1. *near*	1. *r*
15	*near*	–	–
16	*o*	1. *on*	1. *n*
17	*on*	1. *onc;* 2. *one*	1. *c*; 2. *e*
18	*onc*	1. *once*	1. *e*
19	*once*	–	–
20	*one*	–	–

Пусть $\tilde{s} = \alpha_{k+1}\tilde{P}_k\alpha_k \ldots \alpha_2\tilde{P}_1\alpha_1\tilde{P}_0\alpha_0$ — зеркальное отражение строки *s*.

Алгоритм построения функции ошибок *failure*

Здесь P — множество строк образцов.

$FalseSuff(P)$

1. $\tilde{s} \leftarrow \alpha_{k+1}\tilde{P}_k\alpha_k \ldots \alpha_2\tilde{P}_1\alpha_1\tilde{P}_0\alpha_0$
2. $p_{\tilde{s}} \leftarrow SuffArr(\tilde{s})$
3. $alpha \leftarrow \{\alpha_1, \alpha_2, \ldots, \alpha_k, \alpha_{k+1}\}$
4. ordered_list $\leftarrow Adaptation(\tilde{s}, p_{\tilde{s}}, alpha)$
5. $link \leftarrow \varnothing$
6. $for\ (i \leftarrow 0; i < |\tilde{s}|; i{+}{+})$
7. $inLink[i] \leftarrow \varepsilon$
8. sub $\leftarrow DandC(ordered_list)$
9. $str \leftarrow \varnothing$
10. $for\ (i \leftarrow 0; i < |sub| - 1; i{+}{+})$
11. $for\ (j \leftarrow sub[i]; j < sub[i+1] - 1; j{+}{+})$
12. $str[j] \leftarrow |lcp(ordered_list[j], ordered_list[j+1])|$
13. $for\ (i \leftarrow 0; i < |sub| - 1; i{+}{+})$
14. $for\ (k \leftarrow sub[i+1] - 1; k > sub[i]; k{-}{-})$ {
15. $for\ (j \leftarrow sub[i]; j < k; j{+}{+})$
16. $min_element \leftarrow \min(str[k-1], str[k-2], \ldots, str[j])$
17. $if\ (\min_element = |ordered_list[j]|)$
18. $\min_temp[j - sub[i]] \leftarrow min_element$
19. }
20. $max_element \leftarrow \max(\min_temp[0], \min_temp[1], \ldots, \min_temp[w])$
21. найти *max_index*: $\min_temp[\max_index] = \max_element$
22. $inLink[k] \leftarrow ordered_list[\max_index + sub[i]]$
23. }
24. $for\ (i \leftarrow 0; i < |inLink|; i{+}{+})$ {

25. $link[i][0] \leftarrow ordered_list[i]^{\sim}$; //зеркальное отражение строки

26. $link[i][1] \leftarrow inLink[i]^{\sim}$; //зеркальное отражение строки

27. }

28. *return link*

Замечание. В строке 20 $w \le sub[i+1]-sub[i]-1$.

Теорема 2. Алгоритм *FalseSuff* строит функцию ошибок *failure* за время $O(n)$. Доказательство. На шаге 1 строим массив символов, содержащий зеркальные отображения строк, принадлежащих множеству образцов *P*, и некоторые уникальные символы. На шаге 2 строим суффиксный массив $p_{\tilde{s}}$ для строки $\tilde{s}$. На шаге 4 при помощи алгоритма *Adaptation* в массив *ordered_list* записываются все суффиксы строк, принадлежащих множеству образцов $\tilde{P}$ (множество образцов, состоящее из зеркально отраженных строк *P*), при этом не исключены повторения. На шаге 8 выполняется алгоритм *DandC* (см. лемму 2), после чего в цикле 10–12 находим длину наибольшего общего префикса между строками, у которых совпадает первый символ. Записываем результат в массив *str*. Заметим, что эта величина равна нулю у строк, для которых данное условие не выполняется. В цикле 13–23 строится специальное отображение между строками, у которых совпадает первый символ. Опишем это отображение. Зафиксируем некоторую строку

$$s \in ordered_list\left[ordered_list\left[sub[i]\right],...,ordered_list\left[sub[i+1]-1\right]\right].$$

Рассмотрим множество, состоящее из строк, принадлежащих $ordered_list\left[ordered_list\left[sub[i]\right],...,ordered_list\left[sub[i+1]-1\right]\right]$. Их длина равна длине наибольшего общего префикса с *s*, исключая само *s*. Из этого множества найдем строку s', которая имеет наибольшую длину, и поставим ее в соответствие *s*. Очевидно, что построенное отображение является биекцией при условии $s' \ne \varepsilon$. Полученный результат записывается в массив *inLink*. В цикле 24–27 при помощи массива *inLink* явно указываем построенное отображение, при этом зеркально отражая каждую из строк. Таким образом, ставим узлу $\tilde{s}$ префиксного дерева, построенного на основе массива образцов *P*, в соответствие узел $\tilde{s}'$. Его состояние равно наибольшему собственному суффиксу $\tilde{s}$, который встречается среди множества состояний рассматриваемого префиксного дерева. Но согласно определению функции ошибок *failure* это и есть искомый результат.

Пусть $n = \sum_{i=1}^{k} |P_i|$. Шаг 2 выполняется за время $O(n+k+1)$ [12]. Из леммы 1 следует, что шаг 4 выполняется за время $O(n+k+1-k-1) = O(n)$. Цикл 6–7 выполняется за время $O(|\tilde{s}|) = O(n)$. Из леммы 2 следует, что шаг 8 выполняется за время $O(n)$. Цикл 10–12 выполняется за время $O\left(\sum_{i=0}^{|sub|-2} \sum_{j=sub[i]+1}^{sub[i+1]-1} \gamma_j\right) = O(k)$, $\forall j \;\; \gamma_j = 1$. Цикл 14–23 выполняется за время

$$O\left(\sum_{k=sub[i]+1}^{sub[i+1]-1} \sum_{j=sub[i]}^{k-1} \gamma_j\right) = O\left(\sum_{j=0}^{sub[i+1]-sub[i]-2} \gamma_j\right) = O\left(sub[i+1]-sub[i]-1\right), \; \forall j \;\; \gamma_j = 1.$$

Тогда цикл 13–23 выполняется за время

$$O\left(\sum_{i=0}^{|sub|-2} \left(sub[i+1]-sub[i]-1\right)\right) = O\left(sub\left[|sub|-1\right]\right) = O(k).$$

Поскольку $|inLink| < n$, то цикл 24–27 выполняется за время $O(|inLink|) = O(n)$. Итак, учитывая, что $k \le n$, получаем асимптотическую оценку времени работы алгоритма $O(n) + O(n+k+1) + O(k) = O(n)$. Теорема доказана.

Пример 2.

Пусть, как и в примере 1, $P = \{one, on, once, cell, lull, eye, near\}$. Тогда

$$\tilde{s} = \alpha_8 raen \alpha_7 eye \alpha_6 llul \alpha_5 llec \alpha_4 ecno \alpha_3 no \alpha_2 eno \alpha_1. \quad (2)$$

В табл. 2 представлен результат работы алгоритма построения функции *failure* при поступлении на вход строки $\tilde{s}$ (2).

http://vestnik.donstu.ru

Таблица 2

Ложные связи между узлами

Массив *inLink*	Массив *link*	Массив *inLink*	Массив *link*
0. ε	0. 0. *nea*; 1. ε	10. *l*	10. 0. *cel*; 1. *l*
1. ε	1. 0. *c*; 1. ε	11. *l*	11. 0. *cell*; 1. *l*
2. *c*	2. 0. *onc*; 1. *c*	12. *l*	12. 0. *lull*; 1. *l*
3. ε	3. 0. *e*; 1. ε	13. *l*	13. 0. *lul*; 1. *l*
4. *e*	4. 0. *ce*; 1. *e*	14. ε	14. 0. *n*; 1. ε
5. *ec*	5. 0. *once*; 1. *ce*	15. *n*	15. 0. *on*; 1. *n*
6. *e*	6. 0. *ne*; 1. *e*	16. ε	16. 0. *o*; 1. ε
7. *en*	7. 0. *one*; 1. *ne*	17. ε	17. 0. *near*; 1. ε
8. *e*	8. 0. *eye*; 1. *e*	18. ε	18. 0. *lu*; 1. ε
9. ε	9. 0. *l*; 1. ε	19. ε	19. 0. *ey*; 1. ε

Для всех узлов, для которых на рис. 1 не указаны ложные связи, полагаем, что ложная связь ведет к корневому узлу.

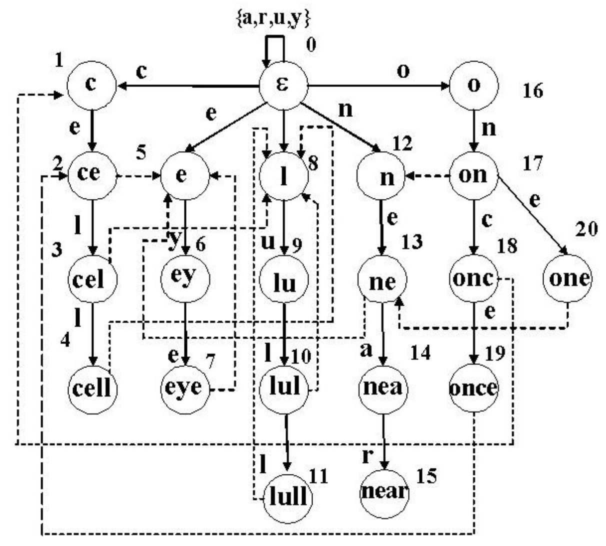


Рис. 1. Префиксное дерево с ложными связями

Обсуждение и заключения. Описана новая процедура препроцессинга в алгоритме Ахо — Корасик. Она выполняется за линейное время $O(n)$. Исследована связь между суффиксными массивами и префиксным деревом, что позволило предложить иной способ построения функций перехода и ошибок. Полученные результаты позволяют сократить время на предвыборную обработку множества строк образцов при использовании целочисленного алфавита.

Библиографический список

1. Stallings, W. Computer security: principles and practice / W. Stallings. — Boston : Pearson, 2012. — 182 p.

2. Исследование возможности применения генетических алгоритмов для реализации криптоанализа блочных криптосистем / Ю. О. Чернышев [и др.] // Вестник Дон. гос. техн. ун-та. — 2015. — Т. 15, № 3 (82). — С. 65–72.

3. Садовой, Н. Н. Программные утилиты для контроля и предотвращения сетевых атак на уровне доступа к сети / Н. Н. Садовой, Ю. В. Косолапов, В. В. Мкртичян // Вестник Дон. гос. техн. ун-та. — 2005. — Т. 5, № 2 (24). — С. 173–178.

4. Могилевская, Н. С. Пороговое разделение файлов на основе битовых масок: идея и возможное применение / Н. С. Могилевская, Р. В. Кульбикаян, Л. А. Журавлев // Вестник Дон. гос. техн. ун-та. — 2011 — Т. 11, № 10. — С. 1749–1755.

5. Шелудько, А. А. Поиск информационных объектов в памяти компьютера при решении задач обеспечения кибербезопасности / А. А. Шелудько, Н. В. Болдырихин // Молодой исследователь Дона. — 2018. — № 6 (15). — С. 81–86.

6. Мазуренко, А. В. Обнаружение, основанное на сигнатурах, с использованием алгоритма Ахо — Корасик / А. В. Мазуренко, Н. В. Болдырихин // Тр. Сев.-Кав. ф-ла Моск. техн. ун-та связи и информатики. — 2016. — № 1. — С. 339–344.

7. Мазуренко, А. В. Алгоритм проверки подлинности пользователя, основанный на графических ключах / А. В. Мазуренко, Н. С. Архангельская, Н. В. Болдырихин // Молодой исследователь Дона. — 2016. — № 3 (3). — С. 92–95.

8. Мазуренко, А. В. Геометрическая реализация метода проведения электронных выборов, основанного на пороговом разделении секрета / А. В. Мазуренко, В. А. Стукопин // Вестник Дон. гос. техн. ун-та. — 2018. — Т. 18, № 2. — С. 246–255.

9. Алгоритмическая оценка сложности системы кодирования и защиты информации, основанной на пороговом разделении секрета, на примере системы электронного голосования / Л. В. Черкесова [и др.] // Вестник Дон. гос. техн. ун-та. — 2017. — Т. 17, № 3. — С. 145–155.

10. Антонов, Е. С. Как найти миллион. Сравнение алгоритмов поиска множества подстрок / Е. С. Антонов // RSDN Magazine. — 2011. — № 1. — С. 60–67.

11. Tarakeswar, M. K. Search Engines: A Study / M. K. Tarakeswar, M. D. Kavitha // Journal of Computer Applications. — 2011. — Vol. 4, № 1. — P. 29–33.

12. Karkkainen, J. Linear work suffix array construction / J. Karkkainen, P. Sanders, S. Burkhardt // Journal of the ACM. — 2006. — Vol. 53, № 6. — P. 918–936.

13. Баклановский, М. В. Поведенческая идентификация программ / М. В. Баклановский, А. Р. Ханов // Моделирование и анализ информационных систем. — 2014. — Т. 21, № 6. — С. 120–130.

14. Efficient repeat finding in sets of strings via suffix arrays / V. Becher [et al.] // Discrete Mathematics and Theoretical Computer Science. — 2013. — Vol. 15, № 2. — P. 59–70.

15. Shrestha, A. M. S. A bioinformatician's guide to the forefront of suffix array construction algorithms / A. M. S. Shrestha, M. C. Frith, P. Horton // Briefings in Bioinformatics. — 2014. — Vol. 15, № 2. — P. 138–154.

Сдана в редакцию 22.01.2019
Принята к публикации 12.04.2019

Об авторах:

Мазуренко Александр Вадимович,
математик-программист ООО «ДДОС-Гвард» (РФ, 344002, г. Ростов-на-Дону, пр. Буденновский, 62/2),
ORCID: http://orcid.org/0000–0001–9541–3374
mazurencoal@gmail.com

Болдырихин Николай Вячеславович,
доцент кафедры «Кибербезопасность информационных систем» Донского государственного технического университета (РФ, 344000, г. Ростов-на-Дону, пл. Гагарина, 1), кандидат технических наук,
ORCID: http://orcid.org/0000–0002–9896–9543,
boldyrikhin@mail.ru